
6.3 Segment-Level Protection

All five aspects of protection apply to segment translation:

1. Type checking
2. Limit checking
3. Restriction of addressable domain
4. Restriction of procedure entry points
5. Restriction of instruction set

The segment is the unit of protection, and segment descriptors store protection parameters. Protection checks are performed automatically by the CPU when the selector of a segment descriptor is loaded into a segment register and with every segment access. Segment registers hold the protection parameters of the currently addressable segments.

6.3.1 Descriptors Store Protection Parameters

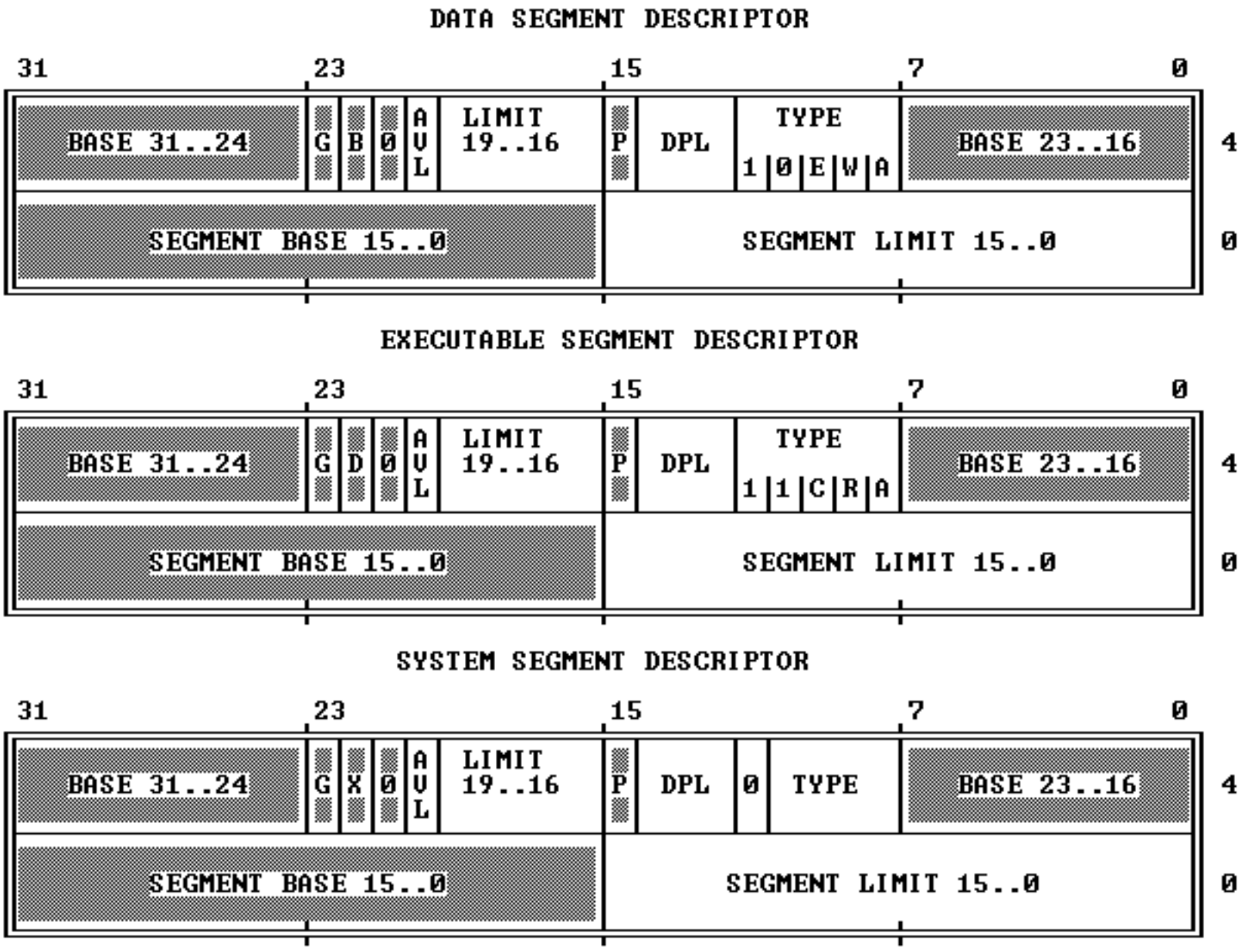
Figure 6-1 highlights the protection-related fields of segment descriptors.

The protection parameters are placed in the descriptor by systems software at the time a descriptor is created. In general, applications programmers do not need to be concerned about protection parameters.

When a program loads a selector into a segment register, the processor loads not only the base address of the

segment but also protection information. Each segment register has bits in the invisible portion for storing base, limit, type, and privilege level; therefore, subsequent protection checks on the same segment do not consume additional clock cycles.

Figure 6-1. Protection Fields of Segment Descriptors



```
A   - ACCESSED                          E   - EXPAND-DOWN
AVL - AVAILABLE FOR PROGRAMMERS USE     G   - GRANULARITY
B   - BIG                               P   - SEGMENT PRESENT
C   - CONFORMING                        R   - READABLE
D   - DEFAULT                           W   - WRITABLE
DPL - DESCRIPTOR PRIVILEGE LEVEL
```

6.3.1.1 Type Checking

The TYPE field of a descriptor has two functions:

1. It distinguishes among different descriptor formats.
2. It specifies the intended usage of a segment.

Besides the descriptors for data and executable segments commonly used by applications programs, the 80386 has descriptors for special segments used by the operating system and for gates. Table 6-1 lists all the types defined for system segments and gates. Note that not all descriptors define segments; gate descriptors have a different purpose that is discussed later in this chapter.

The type fields of data and executable segment descriptors include bits which further define the purpose of the segment (refer to Figure 6-1):

- The writable bit in a data-segment descriptor specifies whether instructions can write into the segment.
- The readable bit in an executable-segment descriptor specifies whether instructions are allowed to read from the segment (for example, to access constants that are stored with instructions). A readable, executable segment may be read in two ways:
    1. Via the CS register, by using a CS override prefix.
    2. By loading a selector of the descriptor into a data-segment register (DS, ES, FS,or GS).

Type checking can be used to detect programming errors that would attempt to use segments in ways not intended by the programmer. The processor examines type information on two kinds of occasions:

1. When a selector of a descriptor is loaded into a segment register. Certain segment registers can contain only certain descriptor types; for example:
    - The CS register can be loaded only with a selector of an executable segment.
    - Selectors of executable segments that are not readable cannot be loaded into data-segment registers.
    - Only selectors of writable data segments can be loaded into SS.
2. When an instruction refers (implicitly or explicitly) to a segment register. Certain segments can be used by instructions only in certain predefined ways; for example:
    - No instruction may write into an executable segment.
    - No instruction may write into a data segment if the writable bit is not set.
    - No instruction may read an executable segment unless the readable bit is set.

Table 6-1. System and Gate Descriptor Types

Code	Type of Segment or Gate
0	-reserved
1	Available 286 TSS
2	LDT
3	Busy 286 TSS
4	Call Gate
5	Task Gate
6	286 Interrupt Gate
7	286 Trap Gate
8	-reserved
9	Available 386 TSS
A	-reserved
B	Busy 386 TSS
C	386 Call Gate
D	-reserved
E	386 Interrupt Gate
F	386 Trap Gate

6.3.1.2 Limit Checking

The limit field of a segment descriptor is used by the processor to prevent programs from addressing outside the segment. The processor's interpretation of the limit depends on the setting of the G (granularity) bit. For data segments, the processor's interpretation of the limit depends also on the E-bit (expansion-direction bit) and the B-bit (big bit) (refer to Table 6-2).

When G=0, the actual limit is the value of the 20-bit limit field as it appears in the descriptor. In this case, the limit may range from 0 to 0FFFFFH (2^(20) - 1 or 1 megabyte). When G=1, the processor appends 12 low-order one-bits to the value in the limit field. In this case the actual limit may range from 0FFFH (2^(12) - 1 or 4 kilobytes) to 0FFFFFFFFH(2^(32) - 1 or 4 gigabytes).

For all types of segments except expand-down data segments, the value of the limit is one less than the size (expressed in bytes) of the segment. The processor causes a general-protection exception in any of these cases:

- Attempt to access a memory byte at an address > limit.
- Attempt to access a memory word at an address >= limit.
- Attempt to access a memory doubleword at an address >= (limit-2).

For expand-down data segments, the limit has the same function but is interpreted differently. In these cases the range of valid addresses is from limit + 1 to either 64K or 2^(32) - 1 (4 Gbytes) depending on the B-bit. An expand-down segment has maximum size when the limit is zero. (Turning on the expand-down bit swaps which bytes are accessible and which are not.)

The expand-down feature makes it possible to expand the size of a stack by copying it to a larger segment without needing also to update intrastack pointers.

The limit field of descriptors for descriptor tables is used by the processor to prevent programs from selecting a table entry outside the descriptor table. The limit of a descriptor table identifies the last valid byte of the last descriptor in the table. Since each descriptor is eight bytes long, the limit value is N * 8 - 1 for a table that can

contain up to N descriptors.

Limit checking catches programming errors such as runaway subscripts and invalid pointer calculations. Such errors are detected when they occur, so that identification of the cause is easier. Without limit checking, such errors could corrupt other modules; the existence of such errors would not be discovered until later, when the corrupted module behaves incorrectly, and when identification of the cause is difficult.

Table 6-2. Useful Combinations of E, G, and B Bits

Case:	1	2	3	4
Expansion Direction	U	U	D	D
G-bit	0	1	0	1
B-bit	X	X	0	1
Lower bound is:	0	0	LIMIT+1	shl(LIMIT,12,1)+1
Upper bound is:	LIMIT	shl(LIMIT,12,1)	64K-1	4G-1
Max seg size is:	64K 0-FFFF	4G 0-FFFFFFFF	64K-1 !(0-0)	4G-4K !(0-FFF)
Min seg size is:	1 0-0	4K 0-FFF	0 !(0-FFFF)	0 !(0-FFFFFFFF)

shl (X, 12, 1) = shift X left by 12 bits inserting one-bits on the right

6.3.1.3 Privilege Levels

The concept of privilege is implemented by assigning a value from zero to three to key objects recognized by the processor. This value is called the privilege level. The value zero represents the greatest privilege, the value three represents the least privilege. The following processor-recognized objects contain privilege levels:

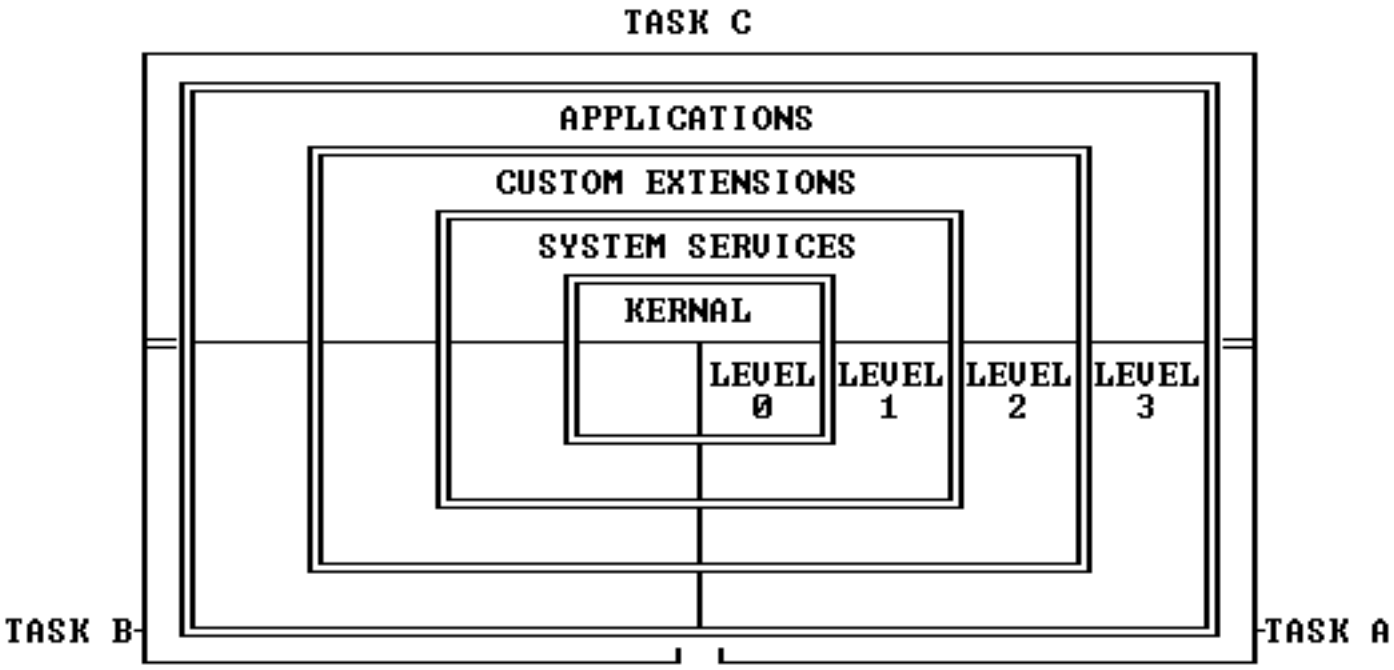
- Descriptors contain a field called the descriptor privilege level (DPL).
- Selectors contain a field called the requestor's privilege level (RPL). The RPL is intended to represent the privilege level of the procedure that originates a selector.
- An internal processor register records the current privilege level (CPL). Normally the CPL is equal to the DPL of the segment that the processor is currently executing. CPL changes as control is transferred to segments with differing DPLs.

The processor automatically evaluates the right of a procedure to access another segment by comparing the CPL to one or more other privilege levels. The evaluation is performed at the time the selector of a descriptor is loaded into a segment register. The criteria used for evaluating access to data differs from that for evaluating transfers of control to executable segments; therefore, the two types of access are considered separately in the following sections.

Figure 6-2 shows how these levels of privilege can be interpreted as rings of protection. The center is for the segments containing the most critical software, usually the kernel of the operating system. Outer rings are for the segments of less critical software.

It is not necessary to use all four privilege levels. Existing software that was designed to use only one or two levels of privilege can simply ignore the other levels offered by the 80386. A one-level system should use privilege level zero; a two-level system should use privilege levels zero and three.

Figure 6-2. Levels of Privilege



6.3.2 Restricting Access to Data

To address operands in memory, an 80386 program must load the selector of a data segment into a data-segment register (DS, ES, FS, GS, SS). The processor automatically evaluates access to a data segment by comparing privilege levels. The evaluation is performed at the time a selector for the descriptor of the target segment is loaded into the data-segment register. As Figure 6-3 shows, three different privilege levels enter into this type of privilege check:

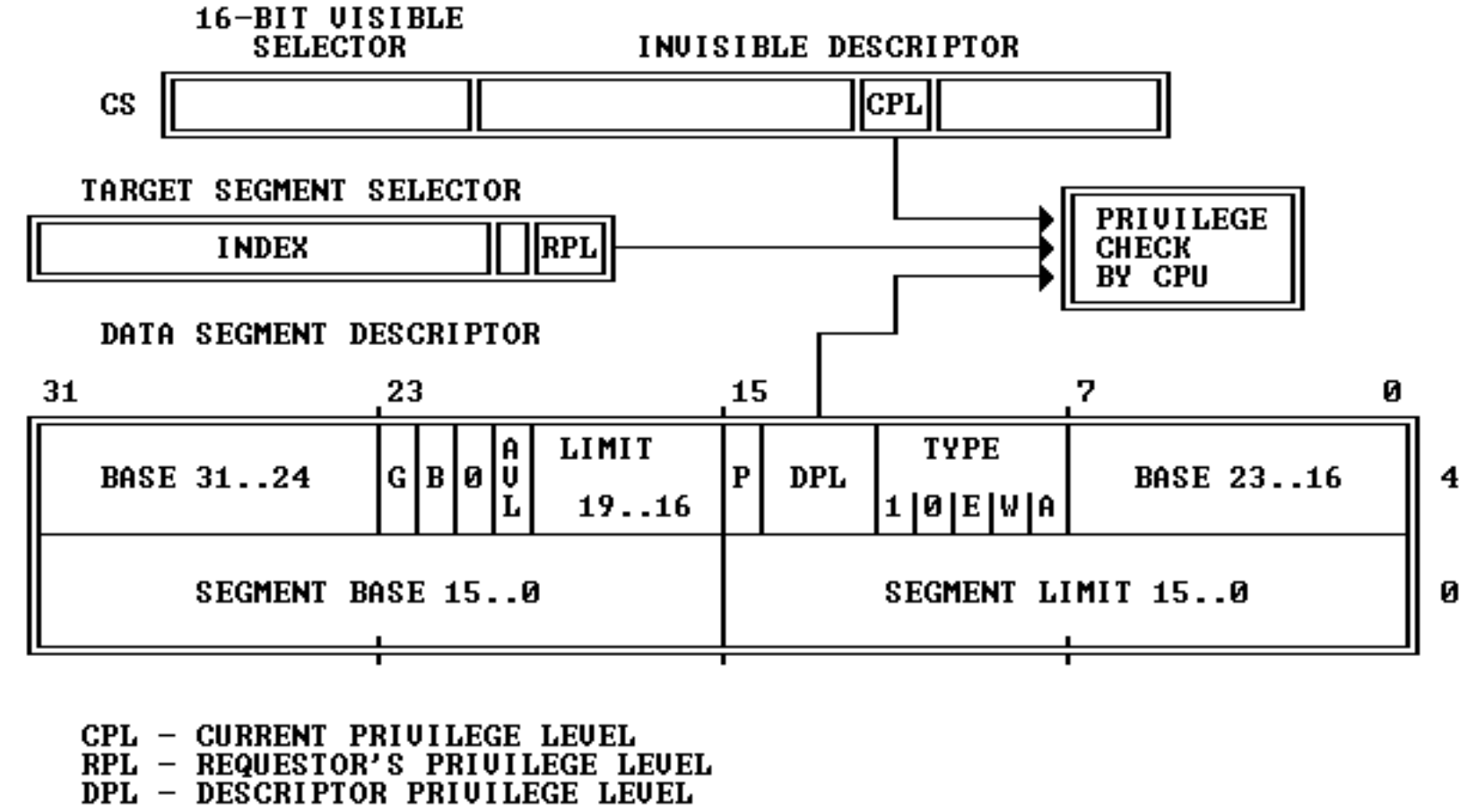
1. The CPL (current privilege level).
2. The RPL (requestor's privilege level) of the selector used to specify the target segment.
3. The DPL of the descriptor of the target segment.

Instructions may load a data-segment register (and subsequently use the target segment) only if the DPL of the

target segment is numerically greater than or equal to the maximum of the CPL and the selector's RPL. In other words, a procedure can only access data that is at the same or less privileged level.

The addressable domain of a task varies as CPL changes. When CPL is zero, data segments at all privilege levels are accessible; when CPL is one, only data segments at privilege levels one through three are accessible; when CPL is three, only data segments at privilege level three are accessible. This property of the 80386 can be used, for example, to prevent applications procedures from reading or changing tables of the operating system.

Figure 6-3. Privilege Check for Data Access



CPL - CURRENT PRIVILEGE LEVEL
RPL - REQUESTOR'S PRIVILEGE LEVEL
DPL - DESCRIPTOR PRIVILEGE LEVEL

6.3.2.1 Accessing Data in Code Segments

Less common than the use of data segments is the use of code segments to store data. Code segments may legitimately hold constants; it is not possible to write to a segment described as a code segment. The following methods of accessing data in code segments are possible:

1. Load a data-segment register with a selector of a nonconforming, readable, executable segment.
2. Load a data-segment register with a selector of a conforming, readable, executable segment.
3. Use a CS override prefix to read a readable, executable segment whose selector is already loaded in the CS register.

The same rules as for access to data segments apply to case 1. Case 2 is always valid because the privilege level of a segment whose conforming bit is set is effectively the same as CPL regardless of its DPL. Case 3 always valid because the DPL of the code segment in CS is, by definition, equal to CPL.

6.3.3 Restricting Control Transfers

With the 80386, control transfers are accomplished by the instructions JMP, CALL, RET, INT, and IRET, as well as by the exception and interrupt mechanisms . Exceptions and interrupts are special cases that Chapter 9 covers. This chapter discusses only JMP, CALL, and RET instructions.

The "near" forms of JMP, CALL, and RET transfer within the current code segment, and therefore are subject only to limit checking. The processor ensures that the destination of the JMP, CALL, or RET instruction does not exceed the limit of the current executable segment. This limit is cached in the CS register; therefore, protection checks for near transfers require no extra clock cycles.

The operands of the "far" forms of JMP and CALL refer to other segments; therefore, the processor performs privilege checking. There are two ways a JMP or CALL can refer to another segment:

1. The operand selects the descriptor of another executable segment.

2. The operand selects a call gate descriptor. This gated form of transfer is discussed in a later section on call gates.

As Figure 6-4 shows, two different privilege levels enter into a privilege check for a control transfer that does not use a call gate:

1. The CPL (current privilege level).
2. The DPL of the descriptor of the target segment.

Normally the CPL is equal to the DPL of the segment that the processor is currently executing. CPL may, however, be greater than DPL if the conforming bit is set in the descriptor of the current executable segment. The processor keeps a record of the CPL cached in the CS register; this value can be different from the DPL in the descriptor of the code segment.

The processor permits a JMP or CALL directly to another segment only if one of the following privilege rules is satisfied:

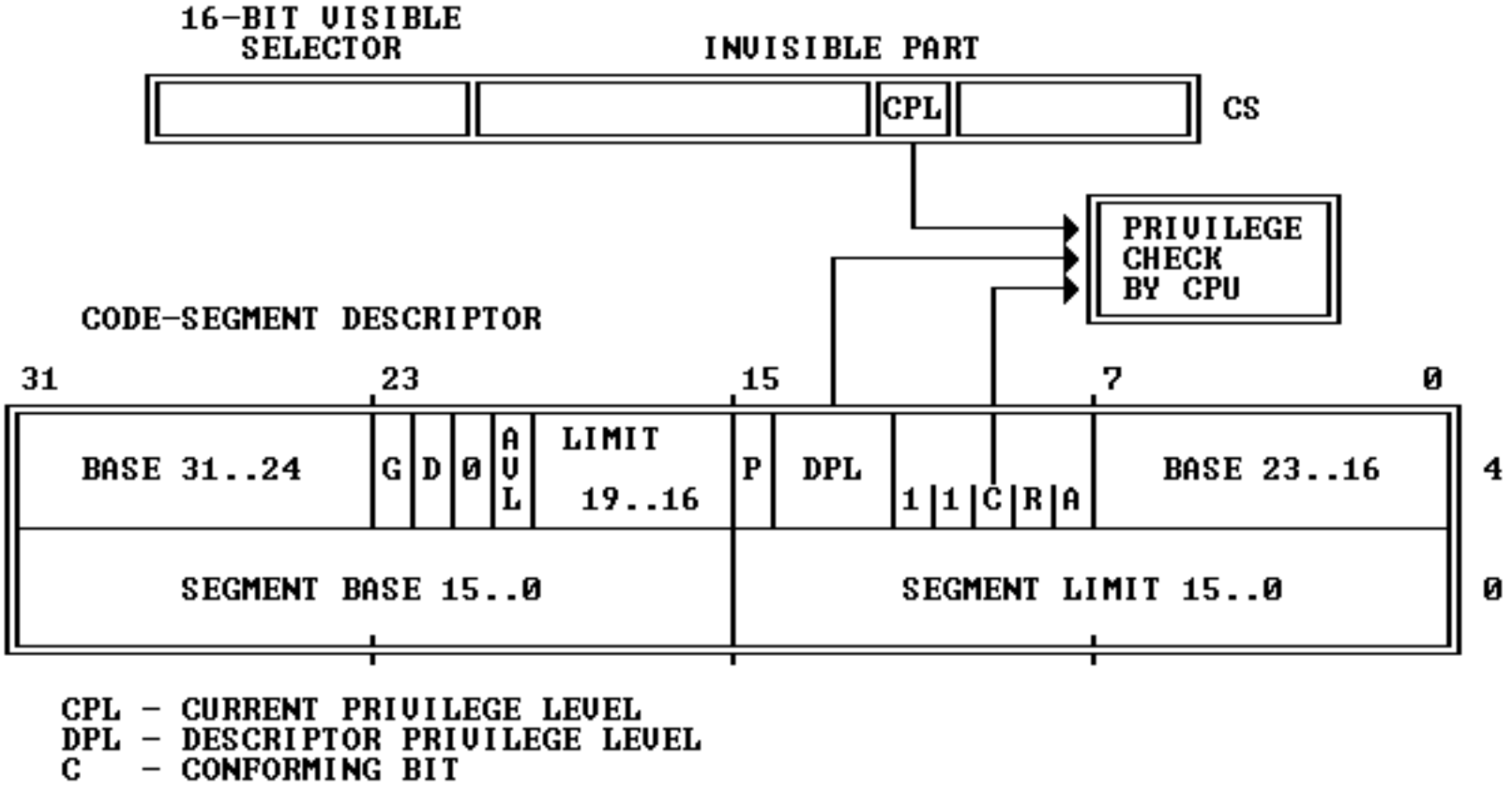
- DPL of the target is equal to CPL.
- The conforming bit of the target code-segment descriptor is set, and the DPL of the target is less than or equal to CPL.

An executable segment whose descriptor has the conforming bit set is called a conforming segment. The conforming-segment mechanism permits sharing of procedures that may be called from various privilege levels but should execute at the privilege level of the calling procedure. Examples of such procedures include math libraries and some exception handlers. When control is transferred to a conforming segment, the CPL does not change. This is the only case when CPL may be unequal to the DPL of the current executable segment.

Most code segments are not conforming. The basic rules of privilege above mean that, for nonconforming segments, control can be transferred without a gate only to executable segments at the same level of privilege. There is a need, however, to transfer control to (numerically) smaller privilege levels; this need is met by the CALL instruction when used with call-gate descriptors, which are explained in the next section. The JMP instruction may never transfer

control to a nonconforming segment whose DPL does not equal CPL.

Figure 6-4. Privilege Check for Control Transfer without Gate



6.3.4 Gate Descriptors Guard Procedure Entry Points

To provide protection for control transfers among executable segments at different privilege levels, the 80386 uses gate descriptors. There are four kinds of gate descriptors:

- Call gates
- Trap gates

- Interrupt gates
- Task gates

This chapter is concerned only with call gates. Task gates are used for task switching , and therefore are discussed in Chapter 7. Chapter 9 explains how trap gates and interrupt gates are used by exceptions and interrupts. Figure 6-5 illustrates the format of a call gate. A call gate descriptor may reside in the GDT or in an LDT, but not in the IDT. A call gate has two primary functions:

1. To define an entry point of a procedure.
2. To specify the privilege level of the entry point.

Call gate descriptors are used by call and jump instructions in the same manner as code segment descriptors. When the hardware recognizes that the destination selector refers to a gate descriptor, the operation of the instruction is expanded as determined by the contents of the call gate.

The selector and offset fields of a gate form a pointer to the entry point of a procedure. A call gate guarantees that all transitions to another segment go to a valid entry point, rather than possibly into the middle of a procedure (or worse, into the middle of an instruction). The far pointer operand of the control transfer instruction does not point to the segment and offset of the target instruction; rather, the selector part of the pointer selects a gate, and the offset is not used. Figure 6-6 illustrates this style of addressing.

As Figure 6-7 shows, four different privilege levels are used to check the validity of a control transfer via a call gate:

1. The CPL (current privilege level).
2. The RPL (requestor's privilege level) of the selector used to specify the call gate.
3. The DPL of the gate descriptor.
4. The DPL of the descriptor of the target executable segment.

The DPL field of the gate descriptor determines what privilege levels can use the gate. One code segment can have several procedures that are intended for use by different privilege levels. For example, an operating system may have some services that are intended to be used by applications, whereas others may be intended only for use by

other systems software.

Gates can be used for control transfers to numerically smaller privilege levels or to the same privilege level (though they are not necessary for transfers to the same level). Only CALL instructions can use gates to transfer to smaller privilege levels. A gate may be used by a JMP instruction only to transfer to an executable segment with the same privilege level or to a conforming segment.

For a JMP instruction to a nonconforming segment, both of the following privilege rules must be satisfied; otherwise, a general protection exception results.

```
MAX (CPL,RPL) <= gate DPL
target segment DPL = CPL
```

For a CALL instruction (or for a JMP instruction to a conforming segment), both of the following privilege rules must be satisfied; otherwise, a general protection exception results.

```
MAX (CPL,RPL) <= gate DPL
target segment DPL <= CPL
```

Figure 6-5. Format of 80386 Call Gate

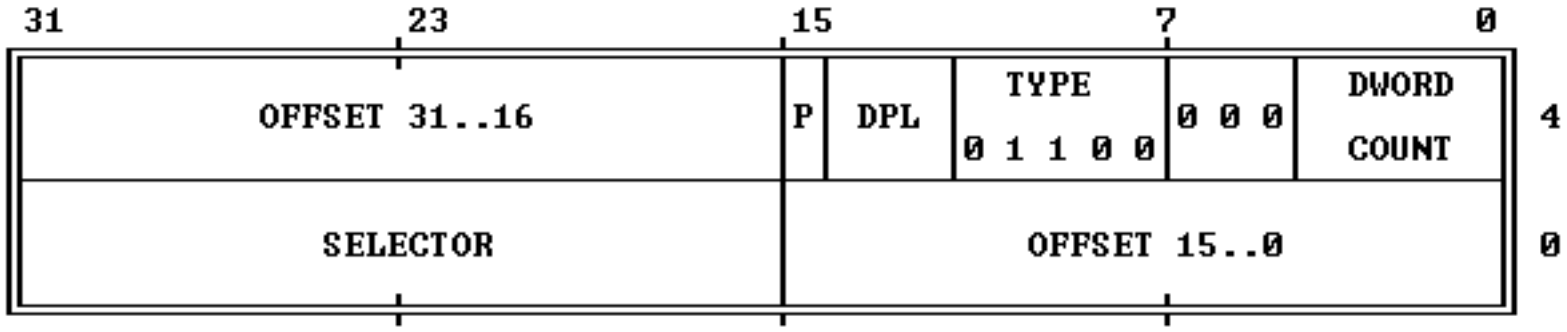


Figure 6-6. Indirect Transfer via Call Gate

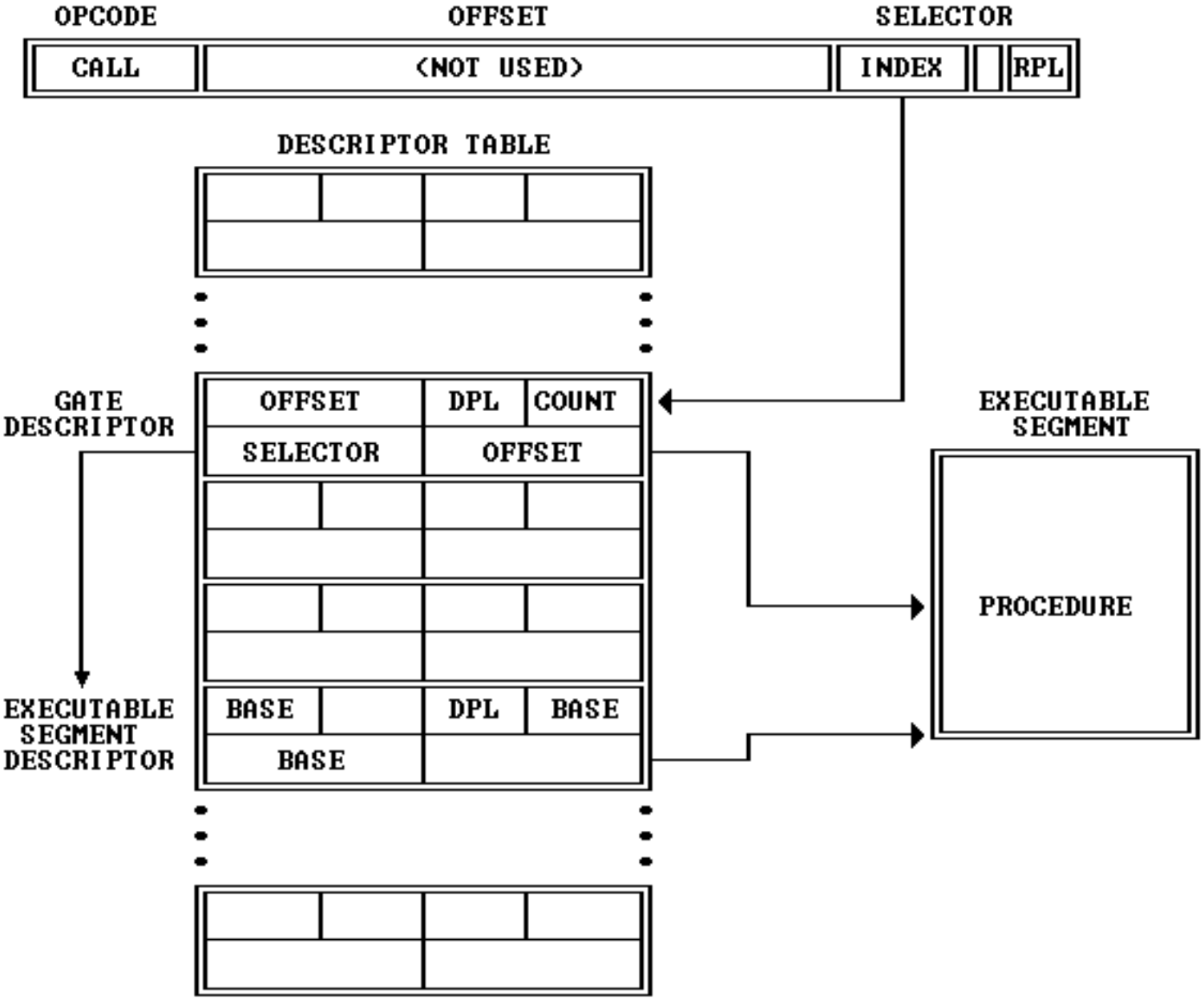
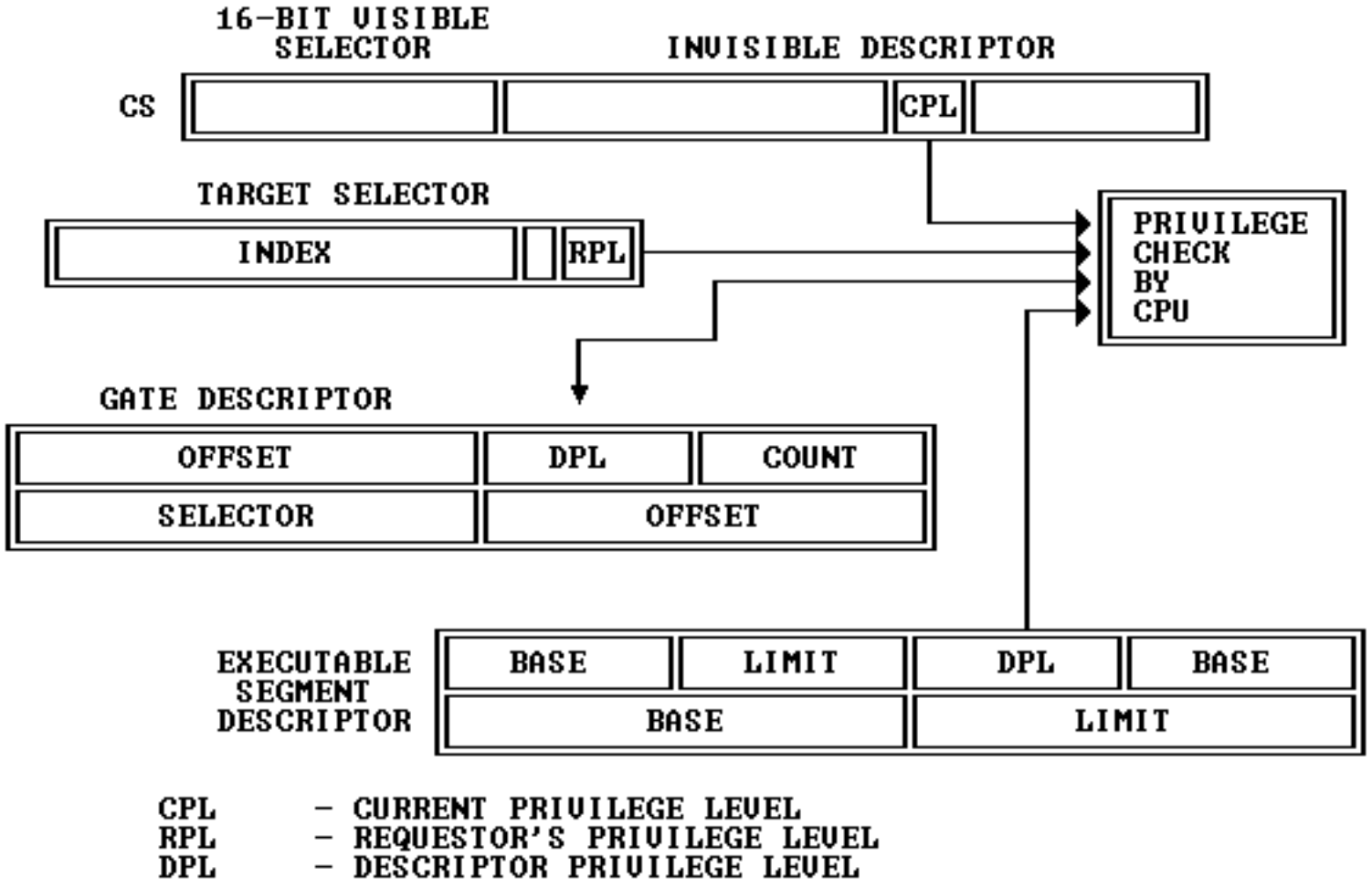


Figure 6-7. Privilege Check via Call Gate



6.3.4.1 Stack Switching

If the destination code segment of the call gate is at a different privilege level than the CPL, an interlevel transfer is being requested.

To maintain system integrity, each privilege level has a separate stack. These stacks assure sufficient stack space to process calls from less privileged levels. Without them, a trusted procedure would not work correctly if the calling

procedure did not provide sufficient space on the caller's stack.

The processor locates these stacks via the task state segment (see Figure 6-8). Each task has a separate TSS, thereby permitting tasks to have separate stacks. Systems software is responsible for creating TSSs and placing correct stack pointers in them. The initial stack pointers in the TSS are strictly read-only values. The processor never changes them during the course of execution.

When a call gate is used to change privilege levels, a new stack is selected by loading a pointer value from the Task State Segment (TSS). The processor uses the DPL of the target code segment (the new CPL) to index the initial stack pointer for PL 0, PL 1, or PL 2.

The DPL of the new stack data segment must equal the new CPL; if it does not, a stack exception occurs. It is the responsibility of systems software to create stacks and stack-segment descriptors for all privilege levels that are used. Each stack must contain enough space to hold the old SS:ESP, the return address, and all parameters and local variables that may be required to process a call.

As with intralevel calls, parameters for the subroutine are placed on the stack. To make privilege transitions transparent to the called procedure, the processor copies the parameters to the new stack. The count field of a call gate tells the processor how many doublewords (up to 31) to copy from the caller's stack to the new stack. If the count is zero, no parameters are copied.

The processor performs the following stack-related steps in executing an interlevel CALL.

1. The new stack is checked to assure that it is large enough to hold the parameters and linkages; if it is not, a stack fault occurs with an error code of 0.
2. The old value of the stack registers SS:ESP is pushed onto the new stack as two doublewords.
3. The parameters are copied.
4. A pointer to the instruction after the CALL instruction (the former value of CS:EIP) is pushed onto the new stack. The final value of SS:ESP points to this return pointer on the new stack.

Figure 6-9 illustrates the stack contents after a successful interlevel call.

The TSS does not have a stack pointer for a privilege level 3 stack, because privilege level 3 cannot be called by any procedure at any other privilege level.

Procedures that may be called from another privilege level and that require more than the 31 doublewords for parameters must use the saved SS:ESP link to access all parameters beyond the last doubleword copied.

A call via a call gate does not check the values of the words copied onto the new stack. The called procedure should check each parameter for validity. A later section discusses how the ARPL, VERR, VERW, LSL, and LAR instructions can be used to check pointer values.

Figure 6-8. Initial Stack Pointers of TSS

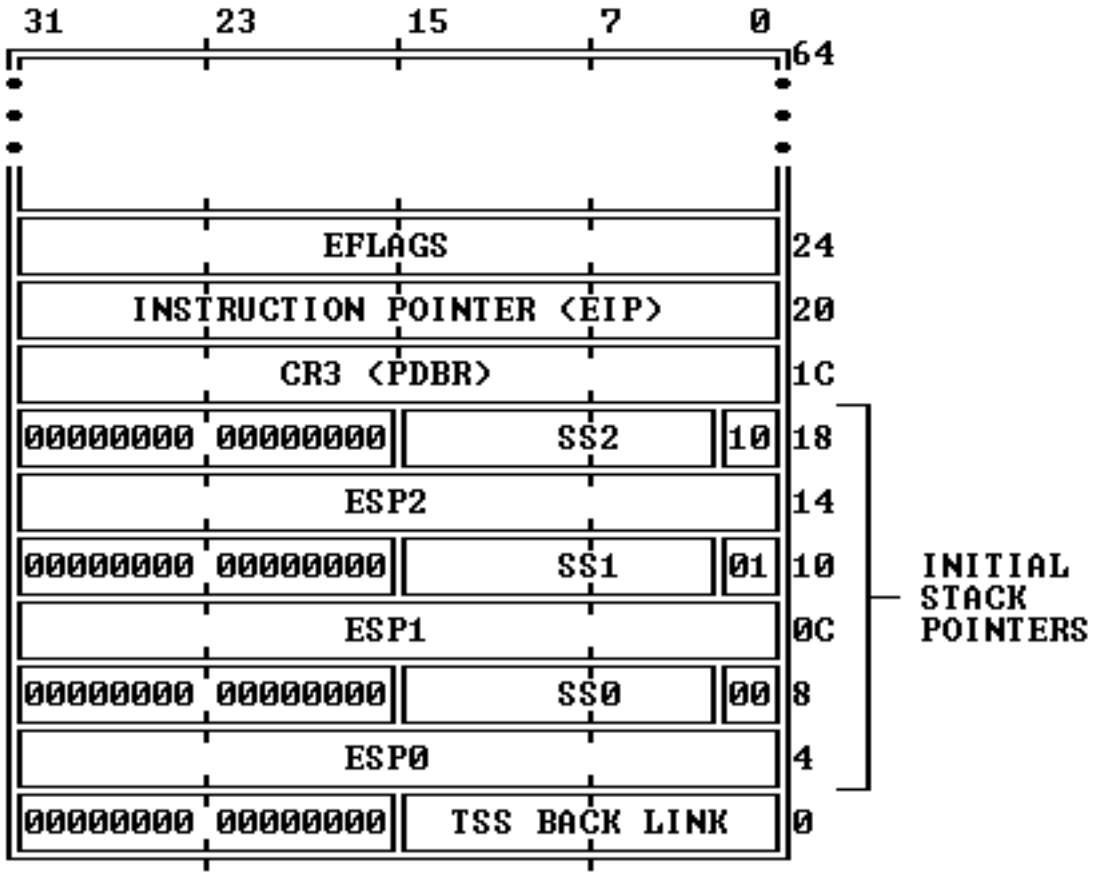
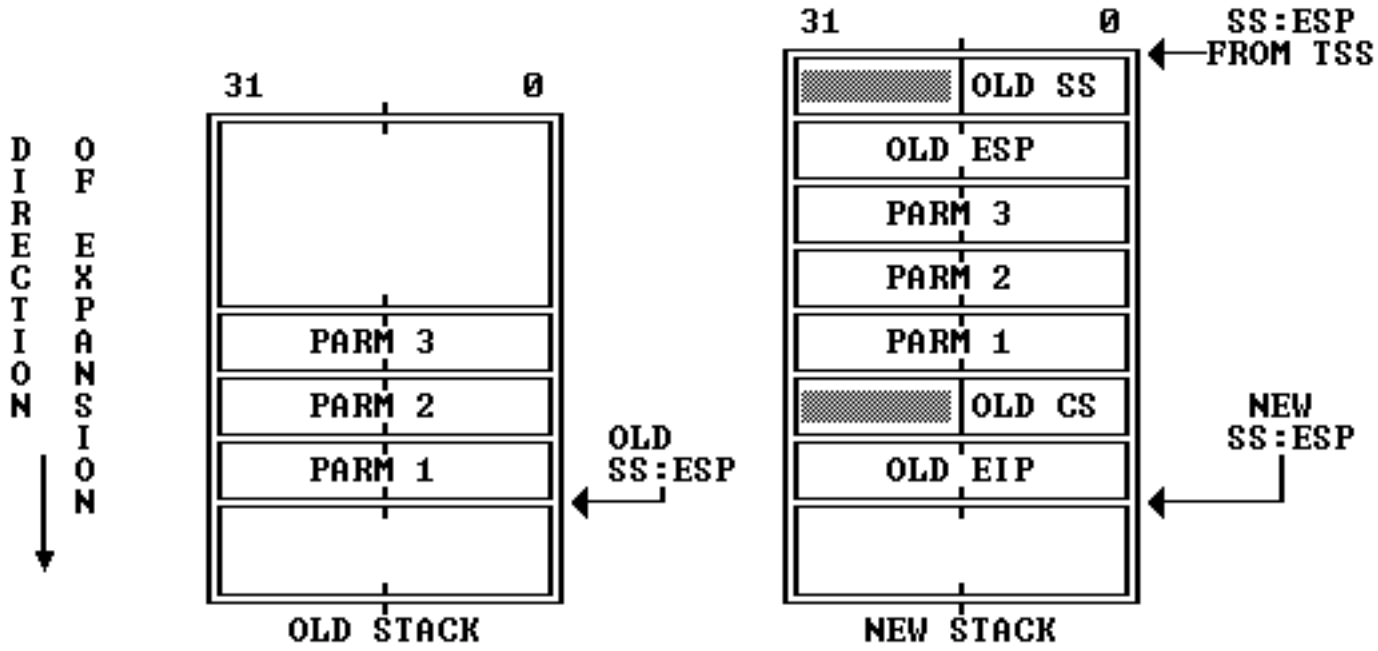


Figure 6-9. Stack Contents after an Interlevel Call



6.3.4.2 Returning from a Procedure

The "near" forms of the RET instruction transfer control within the current code segment and therefore are subject only to limit checking. The offset of the instruction following the corresponding CALL, is popped from the stack. The processor ensures that this offset does not exceed the limit of the current executable segment.

The "far" form of the RET instruction pops the return pointer that was pushed onto the stack by a prior far CALL instruction. Under normal conditions, the return pointer is valid, because of its relation to the prior CALL or INT. Nevertheless, the processor performs privilege checking because of the possibility that the current procedure altered the pointer or failed to properly maintain the stack. The RPL of the CS selector popped off the stack by the return instruction identifies the privilege level of the calling procedure.

An intersegment return instruction can change privilege levels, but only toward procedures of lesser privilege. When the RET instruction encounters a saved CS value whose RPL is numerically greater than the CPL, an interlevel return occurs. Such a return follows these steps:

1. The checks shown in Table 6-3 are made, and CS:EIP and SS:ESP are loaded with their former values that were saved on the stack.
2. The old SS:ESP (from the top of the current stack) value is adjusted by the number of bytes indicated in the RET instruction. The resulting ESP value is not compared to the limit of the stack segment. If ESP is beyond the limit, that fact is not recognized until the next stack operation. (The SS:ESP value of the returning procedure is not preserved; normally, this value is the same as that contained in the TSS.)
3. The contents of the DS, ES, FS, and GS segment registers are checked. If any of these registers refer to segments whose DPL is greater than the new CPL (excluding conforming code segments), the segment register is loaded with the null selector (INDEX = 0, TI = 0). The RET instruction itself does not signal exceptions in these cases; however, any subsequent memory reference that attempts to use a segment register that contains the null selector will cause a general protection exception. This prevents less privileged code from accessing more privileged segments using selectors left in the segment registers by the more privileged procedure.

6.3.5 Some Instructions are Reserved for Operating System

Instructions that have the power to affect the protection mechanism or to influence general system performance can only be executed by trusted procedures. The 80386 has two classes of such instructions:

1. Privileged instructions -- those used for system control.
2. Sensitive instructions -- those used for I/O and I/O related activities.

```
Table 6-3. Interlevel Return Checks

SF = Stack Fault
GP = General Protection Exception
NP = Segment-Not-Present Exception
```

Type of Check	Exception	Error Code
ESP is within current SS segment	SF	0
ESP + 7 is within current SS segment	SF	0
RPL of return CS is greater than CPL	GP	Return CS
Return CS selector is not null	GP	Return CS
Return CS segment is within descriptor table limit	GP	Return CS
Return CS descriptor is a code segment	GP	Return CS
Return CS segment is present	NP	Return CS
DPL of return nonconforming code segment = RPL of return CS, or DPL of return conforming code segment <= RPL of return CS	GP	Return CS
ESP + N + 15 is within SS segment		
N Immediate Operand of RET N Instruction	SF	Return SS
SS selector at ESP + N + 12 is not null	GP	Return SS
SS selector at ESP + N + 12 is within descriptor table limit	GP	Return SS
SS descriptor is writable data segment	GP	Return SS
SS segment is present	SF	Return SS
Saved SS segment DPL = RPL of saved CS	GP	Return SS
Saved SS selector RPL = Saved SS segment DPL	GP	Return SS

6.3.5.1 Privileged Instructions

The instructions that affect system data structures can only be executed when CPL is zero. If the CPU encounters one of these instructions when CPL is greater than zero, it signals a general protection exception. These instructions include:

- CLTS -- Clear Task-Switched Flag
- HLT -- Halt Processor

- LGDT -- Load GDL Register
- LIDT -- Load IDT Register
- LLDT -- Load LDT Register
- LMSW -- Load Machine Status Word
- LTR -- Load Task Register
- MOV to/from CRn -- Move to Control Register n
- MOV to /from DRn -- Move to Debug Register n
- MOV to/from TRn -- Move to Test Register n

6.3.5.2 Sensitive Instructions

Instructions that deal with I/O need to be restricted but also need to be executed by procedures executing at privilege levels other than zero. The mechanisms for restriction of I/O operations are covered in detail in Chapter 8, "Input/Output".

6.3.6 Instructions for Pointer Validation

Pointer validation is an important part of locating programming errors. Pointer validation is necessary for maintaining isolation between the privilege levels. Pointer validation consists of the following steps:

1. Check if the supplier of the pointer is entitled to access the segment.
2. Check if the segment type is appropriate to its intended use.
3. Check if the pointer violates the segment limit.

Although the 80386 processor automatically performs checks 2 and 3 during instruction execution, software must assist in performing the first check. The unprivileged instruction ARPL is provided for this purpose. Software can also explicitly perform steps 2 and 3 to check for potential violations (rather than waiting for an exception). The unprivileged instructions LAR, LSL, VERR, and VERW are provided for this purpose.

LAR (Load Access Rights) is used to verify that a pointer refers to a segment of the proper privilege level and type. LAR has one operand selector for a descriptor whose access rights are to be examined. The descriptor must be visible at the privilege level which is the maximum of the CPL and the selector's RPL. If the descriptor is visible, LAR obtains a masked form of the second doubleword of the descriptor, masks this value with 00FxFF00H, stores the result into the specified 32-bit destination register, and sets the zero flag. (The x indicates that the corresponding four bits of the stored value are undefined.) Once loaded, the access-rights bits can be tested. All valid descriptor types can be tested by the LAR instruction. If the RPL or CPL is greater than DPL, or if the selector is outside the table limit, no access-rights value is returned, and the zero flag is cleared. Conforming code segments may be accessed from any privilege level.

LSL (Load Segment Limit) allows software to test the limit of a descriptor. If the descriptor denoted by the given selector (in memory or a register) is visible at the CPL, LSL loads the specified 32-bit register with a 32-bit, byte granular, unscrambled limit that is calculated from fragmented limit fields and the G-bit of that descriptor. This can only be done for segments (data, code, task state, and local descriptor tables); gate descriptors are inaccessible. (Table 6-4 lists in detail which types are valid and which are not.) Interpreting the limit is a function of the segment type. For example, downward expandable data segments treat the limit differently than code segments do. For both LAR and LSL, the zero flag (ZF) is set if the loading was performed; otherwise, the ZF is cleared.

Table 6-4. Valid Descriptor Types for LSL

Type Code	Descriptor Type	Valid?
0	(invalid)	NO
1	Available 286 TSS	YES
2	LDT	YES
3	Busy 286 TSS	YES
4	286 Call Gate	NO
5	Task Gate	NO
6	286 Trap Gate	NO
7	286 Interrupt Gate	NO
8	(invalid)	NO
9	Available 386 TSS	YES

```
A       (invalid)                   NO
B       Busy 386 TSS                YES
C       386 Call Gate               NO
D       (invalid)                   NO
E       386 Trap Gate               NO
F       386 Interrupt Gate          NO
```

6.3.6.1 Descriptor Validation

The 80386 has two instructions, VERR and VERW, which determine whether a selector points to a segment that can be read or written at the current privilege level. Neither instruction causes a protection fault if the result is negative.

VERR (Verify for Reading) verifies a segment for reading and loads ZF with 1 if that segment is readable from the current privilege level. VERR checks that:

- The selector points to a descriptor within the bounds of the GDT or LDT.
- It denotes a code or data segment descriptor.
- The segment is readable and of appropriate privilege level.

The privilege check for data segments and nonconforming code segments is that the DPL must be numerically greater than or equal to both the CPL and the selector's RPL. Conforming segments are not checked for privilege level.

VERW (Verify for Writing) provides the same capability as VERR for verifying writability. Like the VERR instruction, VERW loads ZF if the result of the writability check is positive. The instruction checks that the descriptor is within bounds, is a segment descriptor, is writable, and that its DPL is numerically greater or equal to both the CPL and the selector's RPL. Code segments are never writable, conforming or not.

6.3.6.2 Pointer Integrity and RPL

The Requestor's Privilege Level (RPL) feature can prevent inappropriate use of pointers that could corrupt the operation of more privileged code or data from a less privileged level.

A common example is a file system procedure, FREAD (file_id, n_bytes, buffer_ptr). This hypothetical procedure reads data from a file into a buffer, overwriting whatever is there. Normally, FREAD would be available at the user level, supplying only pointers to the file system procedures and data located and operating at a privileged level. Normally, such a procedure prevents user-level procedures from directly changing the file tables. However, in the absence of a standard protocol for checking pointer validity, a user-level procedure could supply a pointer into the file tables in place of its buffer pointer, causing the FREAD procedure to corrupt them unwittingly.

Use of RPL can avoid such problems. The RPL field allows a privilege attribute to be assigned to a selector. This privilege attribute would normally indicate the privilege level of the code which generated the selector. The 80386 processor automatically checks the RPL of any selector loaded into a segment register to determine whether the RPL allows access.

To take advantage of the processor's checking of RPL, the called procedure need only ensure that all selectors passed to it have an RPL at least as high (numerically) as the original caller's CPL. This action guarantees that selectors are not more trusted than their supplier. If one of the selectors is used to access a segment that the caller would not be able to access directly, i.e., the RPL is numerically greater than the DPL, then a protection fault will result when that selector is loaded into a segment register.

ARPL (Adjust Requestor's Privilege Level) adjusts the RPL field of a selector to become the larger of its original value and the value of the RPL field in a specified register. The latter is normally loaded from the image of the caller's CS register which is on the stack. If the adjustment changes the selector's RPL, ZF (the zero flag) is set; otherwise, ZF is cleared.

---

up: Chapter 6 -- Protection
prev: 6.2 Overview of 80386 Protection Mechanisms
next: 6.4 Page-Level Protection